**Лекция  Основные понятия тестирования**

**Основные понятия тестирования**

**Аннотация:**Рассмотрены подходы к обоснованию истинности формул и программ и их связь с тестированием. Представлены на конкретных примерах понятия отладки и тестирования. Рассмотрены вопросы организации тестирования. На примерах пояснены методы поиска ошибок и процедура тестирования. Рассмотрены фазы тестирования, основные проблемы тестирования и поставлена задача выбора конечного набора тестов.

**Ключевые слова:**

программа, суперпозиция, выражение, операторы, истинность формул, отладка, тестирование, фаза тестирования, отображение, компьютер, [testbed](http://www.intuit.ru/studies/courses/48/48/lecture/1426?page=1#keyword26), график, оракул, выходные данные, алгоритм, incident, метод поиска ошибок, пошаговое выполнение, single-stepping, точка останова, дамп, путь, управляющий граф, observation, aspect, ADS, набор тестов, test environment, прогон тестов,test monitoring, test driver, test log, проблема тестирования, определение, множества, поток, нумерация, ветвь, вывод, выбор конечного набора тестов

**Концепция тестирования**

*Программа* – это аналог формулы в обычной математике.

Формула для функции f, полученной *суперпозицией* функций f1, f2, ... fn – *выражение*, описывающее эту *суперпозицию*.

**f = f1\* f2\* f3\*... \* fn**

Если аналог f1,f2,... fn – *операторы* языка программирования, то их формула – *программа*.

Существует два метода обоснования *истинности формул*:

1. **Формальный подход** или **доказательство** применяется, когда из исходных формул-аксиом с помощью формальных процедур (правил вывода) выводятся искомые формулы и утверждения (теоремы). Вывод осуществляется путем перехода от одних формул к другим по строгим правилам, которые позволяют свести процедуру перехода от формулы к формуле к последовательности текстовых подстановок:

**A\*\*3 = A\*A\*A**

**A\*A\*A = A -> R, A\*R -> R, A\*R -> R**

Преимущество формального подхода заключается в том, что с его помощью удается избегать обращений к бесконечной области значений и на каждом шаге доказательства оперировать только конечным множеством символов.

1. **Интерпретационный подход** применяется, когда осуществляется подстановка констант в формулы, а затем интерпретация формул как осмысленных утверждений в элементах множеств конкретных значений. *Истинность* интерпретируемых формул проверяется на конечных множествах возможных значений. Сложность подхода состоит в том, что на конечных множествах комбинации возможных значений для реализации исчерпывающей проверки могут оказаться достаточно велики.

Интерпретационный подход используется при экспериментальной проверке соответствия программы своей спецификации.

Применение интерпретационного подхода в форме экспериментов над исполняемой программой составляет суть *отладки* и *тестирования*.

**Основная терминология**

***Отладка (debug, debugging)*** – процесс поиска, локализации и исправления ошибок в программе [ 9 ] [IEEE Std.610-12.1990].

Термин " *отладка* " в отечественной литературе используется двояко: для обозначения активности по поиску ошибок (собственно *тестирование*), по нахождению причин их появления и исправлению, или активности по локализации и исправлению ошибок.

***Тестирование*** обеспечивает выявление (констатацию наличия) фактов расхождений с требованиями (ошибок).

Как правило, на ***фазе тестирования*** осуществляется и исправление идентифицированных ошибок, включающее локализацию ошибок, нахождение причин ошибок и соответствующую корректировку программы тестируемого приложения (Application Under Testing (AUT) или Implementation Under Testing (IUT)).

Если *программа* не содержит синтаксических ошибок (прошла трансляцию) и может быть выполнена на компьютере, она обязательно вычисляет какую-либо функцию, осуществляющую *отображение* входных данных в выходные. Это означает, что *компьютер* на своих ресурсах доопределяет частично определенную программой функцию до тотальной определенности. Следовательно, судить о правильности или неправильности результатов выполнения программы можно, только сравнивая спецификацию желаемой функции с результатами ее вычисления, что и осуществляется в процессе *тестирования*.

**Пример поиска и исправления ошибки**

***Отладка*** обеспечивает локализацию ошибок, поиск причин ошибок и соответствующую корректировку программы (Пример 2.1, Пример 2.2).

**// Метод вычисляет неотрицательную**

**// степень n числа x**

**static public double Power(double x, int n)**

**{**

**double z=1;**

**for (int i=1;n>=i;i++)**

**{**

**z = z\*x;**

**}**

**return z;**

**}**

**2.1. Исходный текст метода Power**

**double Power(double x,int n)**

**{**

**double z=1;**

**int i;**

**for(i=1;n>=i;i++)**

**{**

**z=z\*x;**

**}**

**return z;**

**}**

**2.1.1. Исходный текст метода Power**

Если вызвать метод Power с отрицательным значением степени n Power(2,-1), то получим некорректный результат 1. Исправим метод так, чтобы ошибочное значение параметра (недопустимое по спецификации значение) идентифицировалось специальным сообщением, а возвращаемый результат был равен 1 (Пример 2.2).

**// Метод вычисляет неотрицательную**

**// степень n числа x**

**static public double PowerNonNeg(double x, int n)**

**{**

**double z=1;**

**if (n>0)**

**{**

**for (int i=1;n>=i;i++)**

**{**

**z = z\*x;**

**}**

**}**

**else Console.WriteLine(**

**"Ошибка ! Степень числа n" +**

**" должна быть больше 0.");**

**return z;**

**}**

**2.2. Скорректированный исходный текст**

**double PowerNonNeg(double x, int n)**

**{**

**double z=1;**

**int i;**

**if (n>0)**

**{**

**for (i=1;n>=i;i++)**

**{**

**z = z\*x;**

**}**

**}**

**else printf("Ошибка! Степень числа n должна быть больше 0.\n");**

**return z;**

**}**

**2.2.1. Скорректированный исходный текст**

Если вызвать скорректированный метод PowerNonNeg(2,-1) с отрицательным значением параметра степени, то сообщение об ошибке будет выдано автоматически.

*Тестирование* разделяют на статическое и динамическое:

Статическое *тестирование* выявляет формальными методами анализа без выполнения тестируемой программы неверные конструкции или неверные отношения объектов программы (ошибки формального задания) с помощью специальных инструментов контроля кода – CodeChecker.

Динамическое *тестирование* (собственно *тестирование*) осуществляет выявление ошибок только на выполняющейся программе с помощью специальных инструментов автоматизации *тестирования* – *Testbed*  [ 9 ] или Testbench.

**Организация тестирования**

Тестирование осуществляется на заданном заранее множестве входных данных X и множестве предполагаемых результатов Y – (X,Y), которые задают график желаемой функции. Кроме того, зафиксирована процедура Оракул (oracle), которая определяет, соответствуют ли выходные данные – Yв (вычисленные по входным данным – X) желаемым результатам – Y, т.е. принадлежит ли каждая вычисленная точка (X,YВ) графику желаемой функции (X,Y).

Оракул дает заключение о факте появления неправильной пары (X,Yв) и ничего не говорит о том, каким образом она была вычислена или каков правильный алгоритм – он только сравнивает вычисленные и желаемые результаты. Оракулом может быть даже Заказчик или программист, производящий соответствующие вычисления в уме, поскольку Оракулу нужен какой-либо альтернативный способ получения функции (X,Y) для вычисления эталонных значений Y.

**Пример сравнения словесного описания пункта спецификации с результатом выполнения фрагмента кода**

Пункт спецификации: "Метод Power должен принимать входные параметры: x – целое число, возводимое в степень, и n – неотрицательный порядок степени. Метод должен возвращать вычисленное значение xn ".

Выполняем метод со следующими параметрами: Power(2,2)

Проверка результата выполнения возможна, когда результат вычисления заранее известен – 4. Если результат выполнения 22 = 4, то он соответствует спецификации.

В процессе тестирования Оракул последовательно получает элементы множества (X,Y) и соответствующие им результаты вычислений (X,Yв) для идентификации фактов несовпадений (test incident).

При выявлении (X,Yв)\notin (X,Y) запускается процедура исправления ошибки, которая заключается во внимательном анализе (просмотре) протокола промежуточных вычислений, приведших к (X,Yв), с помощью следующих методов:

1. "Выполнение программы в уме" (deskchecking).
2. Вставка операторов протоколирования (печати) промежуточных результатов (logging).

**Пример вставки операторов протоколирования промежуточных результатов**

Можно выводить промежуточные значения переменных при выполнении программы. Код, осуществляющий вывод, расположен ниже ([Пример 2.3](http://www.intuit.ru/studies/courses/48/48/lecture/1426?page=2#example.2.3)). Этот метод относится к наиболее популярным средствам автоматизации отладки программистов прошлых десятилетий. В настоящее время он известен как метод внедрения "агентов" в текст отлаживаемой программы.

**// Метод вычисляет неотрицательную**

**// степень n числа x**

**static public double Power(double x, int n)**

**{**

**double z=1;**

**for (int i=1;n>=i;i++)**

**{**

**z = z\*x;**

**Console.WriteLine("i = {0} z = {1}", i, z);**

**}**

**return z;**

**}**

2.3. Исходный текст метода Power со вставкой оператора протоколирования

**double Power(double x, int n)**

**{**

**double z=1;**

**int i;**

**for (i=1;n>=i;i++)**

**{**

**z = z\*x;**

**printf("i = %d z = %f\n",i,z);**

**}**

**return z;**

**}**

2.3.1. Исходный текст метода Power со вставкой оператора протоколирования

1. Пошаговое выполнение программы (single-step running).

**Пример пошагового выполнения программы**

При пошаговом выполнении программы код выполняется строчка за строчкой. В среде Microsoft Visual Studio.NET возможны следующие команды пошагового выполнения:

Step Into – если выполняемая строчка кода содержит вызов функции, процедуры или метода, то происходит вызов, и программа останавливается на первой строчке вызываемой функции, процедуры или метода.

Step Over - если выполняемая строчка кода содержит вызов функции, процедуры или метода, то происходит вызов и выполнение всей функции и программа останавливается на первой строчке после вызываемой функции.

Step Out – предназначена для выхода из функции в вызывающую функцию. Эта команда продолжит выполнение функции и остановит выполнение на первой строчке после вызываемой функции.

Пошаговое выполнение до сих пор является мощным методом автономного тестирования и отладки небольших программ.

Выполнение с заказанными остановками (breakpoints), анализом трасс (traces) или состояний памяти - дампов (dump).

Пример выполнения программы с заказанными контрольными точками и анализом трасс и дампов

Контрольная точка (breakpoint) – точка программы, которая при ее достижении посылает отладчику сигнал. По этому сигналу либо временно приостанавливается выполнение отлаживаемой программы, либо запускается программа "агент", фиксирующая состояние заранее определенных переменных или областей в данный момент.

Когда выполнение в контрольной точке приостанавливается, отлаживаемая программа переходит в режим останова (break mode). Вход в режим останова не прерывает и не заканчивает выполнение программы и позволяет анализировать состояние отдельных переменных или структур данных. Возврат из режима break mode в режим выполнения может произойти в любой момент по желанию пользователя.

Когда в контрольной точке вызывается программа "агент", она тоже приостанавливает выполнение отлаживаемой программы, но только на время, необходимое для фиксации состояния выбранных переменных или структур данных в специальном электронном журнале - Log-файле, после чего происходит автоматический возврат в режим исполнения.

Трасса - это "сохраненный путь " на управляющем графе программы, т.е. зафиксированные в журнале записи о состояниях переменных в заданных точках в ходе выполнения программы.

Например: на [рисунке 2.1](http://www.intuit.ru/studies/courses/48/48/lecture/1426?page=2#image.2.1) условно изображен управляющий граф некоторой программы. Трасса, проходящая через вершины 0-1-3-4-5 зафиксирована в [таблице 2.1](http://www.intuit.ru/studies/courses/48/48/lecture/1426?page=2#table.2.1). Строки таблицы отображают вершины управляющего графа программы, или breakpoints, в которых фиксировались текущие значения заказанных пользователем переменных.

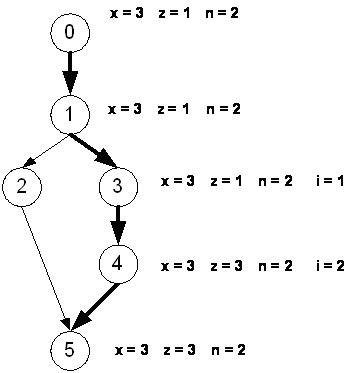


Рисунок 2.1 - Управляющий граф программы

Таблица 2.1 - Трасса, проходящая через вершины 0-1-3-4-5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | | |
| № вершины-оператора | Значение переменной x | Значение переменной z | Значение переменной n | Значение переменной i |
| 0 | 3 | 1 | 2 | не зафиксировано |
| 1 | 3 | 1 | 2 | не зафиксировано |
| 3 | 3 | 1 | 2 | 1 |
| 4 | 3 | 3 | 2 | 2 |
| 5 | 3 | 3 | 2 | не зафиксировано |

Дамп – область памяти, состояние которой фиксируется в контрольной точке в виде единого массива или нескольких связанных массивов. При анализе, который осуществляется после выполнения трассы в режиме off-line, состояния дампа структурируются, и выделенные области или поля сравниваются с состояниями, предусмотренными спецификацией. Например, при моделировании поведения управляющих программ контроллеров в виде дампа фиксируются области общих и специальных регистров, или целые области оперативной памяти, состояния которой определяет алгоритм управления внешней средой.

**Реверсивное (обратное) выполнение (reversible execution)**

Обратное выполнение программы возможно при условии сохранения на каждом шаге программы всех значений переменных или состояний программы для соответствующей трассы. Тогда поднимаясь от конечной точки трассы к любой другой, можно по шагам произвести вычисления состояний, двигаясь от следствия к причине, от состояний на выходе преобразователя данных к состояниям на его входе. Естественно, такие возможности мы получаем в режиме off-line анализа при фиксации в Log – файле всей истории выполнения трассы.

**Пример обратного выполнения для программы вычисления степени числа x**

В программе на Пример 2.4 фиксируются значения всех переменных после выполнения каждого оператора.

**// Метод вычисляет неотрицательную**

**// степень n числа x**

**static public double PowerNonNeg(double x,**

**int n)**

**{**

**double z=1;**

**Console.WriteLine("x={0} z={1} n={2}",**

**x,z,n);**

**if (n>0)**

**{**

**Console.WriteLine("x={0} z={1} n={2}",**

**x,z,n);**

**for (int i=1;n>=i;i++)**

**{**

**z = z\*x;**

**Console.WriteLine(**

**"x={0} z={1} n={2}" +**

**" i={3}",x,z,n,i);**

**}**

**}**

**else Console.WriteLine(**

**"Ошибка ! Степень" +**

**" числа n должна быть больше 0.");**

**return z;**

**}**

2.4. Исходный код с фиксацией результатов выполнения операторов

**double PowerNonNeg(double x, int n)**

**{**

**double z=1;**

**int i;**

**printf("x=%f z=%f n=%d\n",x,z,n);**

**if (n>0)**

**{**

**printf("x=%f z=%f n=%d\n",x,z,n);**

**for (i=1;n>=i;i++)**

**{**

**z = z\*x;**

**printf("x=%f z=%f n=%d i=%d\n",**

**x,z,n,i);**

**}**

**}**

**else printf(**

**"Ошибка ! Степень "**

**"числа n должна быть больше 0.\n");**

**return z;**

**}**

2.4.1. Исходный код с фиксацией результатов выполнения операторов

Зная структуру *управляющего графа* программы и имея значения всех переменных после выполнения каждого оператора, можно осуществить обратное выполнение (например, в уме), подставляя значения переменных в операторы и двигаясь снизу вверх, начиная с последнего.

Итак, в процессе *тестирования* сравнение промежуточных результатов с полученными независимо эталонными результатами позволяет найти причины и место ошибки, исправить текст программы, провести повторную трансляцию и настройку на выполнение и продолжить *тестирование*.

*Тестирование* заканчивается, когда выполнилось или "прошло" (pass) успешно достаточное количество тестов в соответствии с выбранным критерием *тестирования*.

***Тестирование*** – это:

* Процесс выполнения ПО системы или компонента в условиях анализа или записи получаемых результатов с целью проверки (оценки) некоторых свойств тестируемого объекта.

The process of operating a system or component under specified conditions, *observing* or recording the results, and making an evaluation of some *aspect* of the system or component [ 9 ] .

* Процесс анализа пункта требований к ПО с целью фиксации различий между существующим состоянием ПО и требуемым (что свидетельствует о проявлении ошибки) при экспериментальной проверке соответствующего пункта требований.

The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate features of software items [[IEEE Std.610-12.1990], [ 9 ] .

* Контролируемое выполнение программы на конечном множестве тестовых данных и анализ результатов этого выполнения для поиска ошибок [IEEE Std 829-1983].

**Сквозной пример тестирования**

Возьмем несколько отличающуюся от Пример 2.4 программу:

**// Метод вычисляет степень n числа x**

**static public double Power(int x, int n)**

**{**

**int z=1;**

**for (int i=1;n>=i;i++)**

**{**

**z = z\*x;**

**}**

**return z;**

**}**

**[STAThread]**

**static void Main(string[] args)**

**{**

**int x;**

**int n;**

**try**

**{**

**Console.WriteLine("Enter x:");**

**x=Convert.ToInt32(Console.ReadLine());**

**if ((x>=0) & (x<=999))**

**{**

**Console.WriteLine("Enter n:");**

**n=Convert.ToInt32(Console.ReadLine());**

**if ((n>=1) & (n<=100))**

**{**

**Console.WriteLine("The power n" + " of x is {0}", Power(x,n));**

**Console.ReadLine();**

**}**

**else**

**{**

**Console.WriteLine("Error : n " + "must be in [1..100]");**

**Console.ReadLine();**

**}**

**}**

**else**

**{**

**Console.WriteLine("Error : x " + "must be in [0..999]");**

**Console.ReadLine();**

**}**

**}**

**catch (Exception e)**

**{**

**Console.WriteLine("Error : Please enter " + "a numeric argument.");**

**Console.ReadLine();**

**}**

**}**

Пример 2.5. Другой пример вычисления степени числа

**#include <stdio.h>**

**double Power(int x, int n)**

**{**

**int z=1;**

**int i;**

**for (i=1;n>=i;i++)**

**{**

**z = z\*x;**

**}**

**return z;**

**}**

**void main(void)**

**{**

**int x;**

**int n;**

**printf("Enter x:");**

**if(scanf("%d",&x))**

**{**

**if ((x>=0) & (x<=999))**

**{**

**printf("Enter n:");**

**if(scanf("%d",&n)) {**

**if ((n>=1) & (n<=100))**

**{**

**printf("The power n of x is %f\n", Power(x,n));**

**}**

**else**

**{**

**printf("Error : n must be in [1..100]\n");**

**}**

**}**

**else**

**{**

**printf("Error : Please enter a numeric argument\n");**

**}**

**}**

**else**

**{**

**printf("Error : x must be in [0..999]\n");**

**}**

**}**

**else**

**{**

**printf("Error : Please enter a numeric argument\n");**

**}**

**}**

2.5.1. Другой пример вычисления степени числа

Для приведенной программы, вычисляющей степень числа (Пример 2.5), воспроизведем последовательность действий, необходимых для *тестирования*.

**Спецификация программы**

На вход программа принимает два параметра: x - число, n – степень. Результат вычисления выводится на консоль.

Значения числа и степени должны быть целыми.

Значения числа, возводимого в степень, должны лежать в диапазоне – [0..999].

Значения степени должны лежать в диапазоне – [1..100].

Если числа, подаваемые на вход, лежат за пределами указанных диапазонов, то должно выдаваться сообщение об ошибке.

**Разработка тестов**

Определим области эквивалентности входных параметров.

Для x – числа, возводимого в степень, определим классы возможных значений:

1. x < 0 (ошибочное)
2. x > 999 (ошибочное)
3. x - не число (ошибочное)
4. 0 <= x <= 999 (корректное)

Для n – степени числа:

1. n < 1 (ошибочное)
2. n > 100 (ошибочное)
3. n - не число (ошибочное)
4. 1 <= n <= 100 (корректное)

**Анализ тестовых случаев**

1. Входные значения: (x = 2, n = 3) (покрывают классы 4, 8).

Ожидаемый результат: The power n of x is 8.

1. Входные значения: {(x = -1, n = 2),(x = 1000, n = 5)} (покрывают классы 1, 2).

Ожидаемый результат: Error : x must be in [0..999].

1. Входные значения: {(x = 100, n = 0),(x = 100, n = 200)} (покрывают классы 5,6).

Ожидаемый результат: Error : n must be in [1..100].

1. Входные значения: (x = *ADS* n = ASD) (покрывают классы эквивалентности 3, 7).

Ожидаемый результат: Error : Please enter a numeric argument.

1. Проверка на граничные значения:
   1. Входные значения: (x = 999, n = 1).

Ожидаемый результат: The power n of x is 999.

* 1. Входные значения: (x = 0, n = 100).

Ожидаемый результат: The power n of x is 0.

**Выполнение тестовых случаев**

Запустим программу с заданными значениями аргументов.

**Оценка результатов выполнения программы на тестах**

В процессе *тестирования* *Оракул* последовательно получает элементы множества (X,Y) и соответствующие им результаты вычислений YВ. В процессе *тестирования* производится оценка результатов выполнения путем сравнения получаемого результата с ожидаемым.

**Три фазы тестирования**

Реализация *тестирования* разделяется на три этапа:

* Создание *тестового набора (test suite)* путем ручной разработки или автоматической генерации для конкретной среды *тестирования* (*testing environment*).
* *Прогон программы на тестах*, управляемый тестовым монитором (*test monitor*, *test driver* [IEEE Std 829-1983], [[ 9 ]](http://www.intuit.ru/studies/courses/48/48/literature#literature.9)) с получением протокола результатов *тестирования* (*test log*).
* Оценка результатов выполнения программы на *наборе тестов* с целью принятия решения о продолжении или остановке *тестирования*.

Основная *проблема тестирования* - *определение* достаточности *множества* тестов для истинности вывода о правильности реализации программы, а также нахождения *множества* тестов, обладающего этим свойством.

**Простой пример**

Рассмотрим вопросы *тестирования* на примере простой программы (Пример 2.6) на языке С#. Текст этой программы и некоторых других несколько видоизменен с целью сделать иллюстрацию описываемых фактов более прозрачной.

**/\* Функция вычисляет неотрицательную**

**степень n числа x \*/**

**1 double Power(double x, int n){**

**2 double z=1; int i;**

**3 for (i=1;**

**4 n>=i;**

**5 i++)**

**6 {z = z\*x;} /\* Возврат в п.4 \*/**

**7 return z;}**

2.6. Пример простой программы на языке С#

**/\* Функция вычисляет неотрицательную**

**степень n числа x \*/**

**1 double Power(double x, int n){**

**2 double z=1; int i;**

**3 for (i=1;**

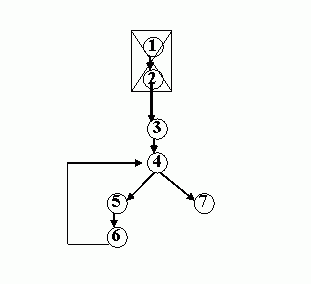
**4 n>=i;**

**5 i++)**

**6 {z = z\*x;} /\* Возврат в п.4 \*/**

**7 return z;}**

2.6.1. Пример простой программы на языке С



**Рисунок 2.2 -**Управляющий граф программы

*Управляющий граф программы (УГП)* на рисунке 2.2 отображает *поток* управления программы. *Нумерация* узлов графа совпадает с нумерацией строк программы. Узлы 1 и 2 не включаются в *УГП*, поскольку отображают строки описаний, т.е. не содержат управляющих операторов.

**Управляющий граф программы**

***Управляющий граф программы (УГП)*** – граф G(V,A), где V(V1,… Vm) – множество вершин (операторов), A(A1,… An) – множество дуг (управлений), соединяющих операторы-вершины.

***Путь*** – последовательность вершин и дуг УГП, в которой любая дуга выходит из вершины Vi и приходит в вершину Vj, например:(3,4,7), (3,4,5,6,4,5,6), (3,4), (3,4,5,6)

***Ветвь*** – *путь* (V1, V2, … Vk), где V1 - либо первый, либо условный оператор программы, Vk - либо условный оператор, либо оператор выхода из программы, а все остальные операторы – безусловные, например: (3,4) (4,5,6,4) (4,7). *Пути*, различающиеся хотя бы числом прохождений цикла – разные *пути*, поэтому число *путей* в программе может быть не ограничено. *Ветви* - линейные участки программы, их конечноe число.

Существуют реализуемые и нереализуемые *пути* в программе, в нереализуемые *пути* в обычных условиях попасть нельзя.

**float H(float x,float y)**

**{**

**float H;**

**1 if (x\*x+y\*y+2<=0)**

**2 H = 17;**

**3 else H = 64;**

**4 return H\*H+x\*x;**

**}**

2.7. Пример описания функции с реализуемыми и нереализуемыми путями

**float H(float x,float y)**

**{**

**float H;**

**1 if (x\*x+y\*y+2<=0)**

**2 H = 17;**

**3 else H = 64;**

**4 return H\*H+x\*x;**

**}**

2.7.1. Пример описания функции с реализуемыми и нереализуемыми путями

Например, для функции Пример 2.7 *путь* (1,3,4) реализуем, *путь* (1,2,4) нереализуем в условиях нормальной работы. Но при сбоях даже нереализуемый *путь* может реализоваться.

**Основные проблемы тестирования**

Рассмотрим два примера *тестирования*:

1. Пусть программа H(x:int, y:int) реализована в машине с 64 разрядными словами, тогда мощность множества тестов ||(X,Y)||=2\*\*128

Это означает, что компьютеру, работающему на частоте 1Ггц, для *прогона* этого *набора тестов* (при условии, что один тест выполняется за 100 команд) потребуется ~ 3K лет.

1. На Рис. 2.3 приведен фрагмент схемы программы управления схватом робота, где интервал между моментами срабатывания схвата не определен.

Этот тривиальный пример требует *прогона* бесконечного множества последовательностей входных значений с разными интервалами срабатывания схвата (Пример 2.8).

**// Прочитать значения датчика**

**static public bool ReadSensor(bool Sensor)**

**{**

**//...чтение значения датчика**

**Console.WriteLine("...reading sensor value");**

**return Sensor;**

**}**

**// Открыть схват**

**static public void OpenHand()**

**{**

**//...открываем схват**

**Console.WriteLine("...opening hand");**

**}**

**// Закрыть схват**

**static public void CloseHand()**

**{**

**//...закрываем схват**

**Console.WriteLine("...closing hand");**

**}**

**[STAThread]**

**static void Main(string[] args)**

**{**

**while (true)**

**{**

**Console.WriteLine("Enter Sensor value (true/false)");**

**if (ReadSensor(Convert.ToBoolean(Console.ReadLine())))**

**{**

**OpenHand();**

**CloseHand();**

**}**

**}**

**}**

**2.8. Фрагмент программы срабатывания схвата**

**#include <stdio.h>**

**/\* Прочитать значения датчика \*/**

**int ReadSensor(int Sensor)**

**{**

**/\* ...чтение значения датчика \*/**

**printf("...reading sensor value\n");**

**return Sensor;**

**}**

**/\* Открыть схват \*/**

**void OpenHand()**

**{**

**/\* ...открываем схват \*/**

**printf("...opening hand\n");**

**}**

**/\* Закрыть схват \*/**

**void CloseHand()**

**{**

**/\* ...закрываем схват \*/**

**printf("...closing hand\n");**

**}**

**void main(void)**

**{**

**int s;**

**while (1)**

**{**

**printf("Enter Sensor value (0/1)");**

**scanf("%d",&s);**

**if (ReadSensor(s))**

**{**

**OpenHand();**

**CloseHand();**

**}**

**}**

**}**

2.8.1. Фрагмент программы срабатывания схвата (рисунок 2.3)



**Рисунок 2.3 -**Тестовая последовательность сигналов датчика схвата

Отсюда *вывод*:

* *Тестирование* программы на всех входных значениях невозможно.
* Невозможно *тестирование* и на всех *путях*.
* Следовательно, надо отбирать конечный *набор тестов*, позволяющий проверить программу **на основе наших интуитивных представлений**

**Требование к тестам** - программа на любом из них должна останавливаться, т.е. **не зацикливаться**. Можно ли заранее гарантировать *останов* на любом тесте?

* В теории алгоритмов доказано, что не существует общего метода для решения этого вопроса, а также вопроса, достигнет ли программа на данном тесте заранее фиксированного оператора.

Задача о *выборе конечного набора тестов* (X,Y) для проверки программы в общем случае неразрешима.

Поэтому для решения практических задач остается искать частные случаи решения этой задачи.

## **Контрольные вопросы**

1. Как можно рассматривать программу с точки зрения математики?
2. Какие два метода обоснования истинности формул существуют?
3. Что такое отладка?
4. Что такое тестирование?
5. Что собой представляет процедура тестирование?
6. В чём назначение Оракула?
7. Что такое статическое тестирование?
8. Что такое динамическое тестирование?
9. Что такое deskchecking?
10. Что такое logging?
11. Что такое УПГ?
12. Что такое путь?
13. Что такое ветвь?
14. Назовите три фазы тестирования?
15. В чём состоит проблема тестирования?
16. В чём состоит требование к тестам?